A Formal Framework for Trace Abstraction and Correlation

Ali Mehrabian

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

December 2010

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Ali Mehrabian

Entitled: "A Formal Framework for Trace Abstraction and Correlation

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. D. Qiu

_____ Examiner, External
Dr. J. Bentahar, CIISE To the Program

_____ Examiner
Dr. A. Agarwal

_____ Supervisor
Dr. A. Hamou-Lhadj

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____2010_____ _____

Dr. Robin A. L. Drew

Dean, Faculty of Engineering and

Computer Science

Abstract

A Formal Framework for Trace Abstraction and Correlation

Ali Mehrabian

Understanding what a software system does and why it does it this way can enable several software engineering tasks including maintenance, performance, and most recently security.

The understanding of software behavior requires some sort of tracing methods. Traces are generated by observing the system during execution. They are then analyzed by users. This analysis, however, is often a tedious task due to the large size of typical traces. Several trace abstraction techniques have been proposed to reduce the impact of this problem. These techniques do not use a formal representation of a trace which hinders the ability to assess their effectiveness. In this thesis we present a formal framework that models trace of routine calls and trace abstraction techniques based on pattern matching. We also present a new method for correlating traces generated from different system using our framework.

The framework is implemented and applied to several traces generated from various object-oriented systems. The results of these case studies are presented in this thesis.

Acknowledgment

During my Masters of Computer Engineering, I had the privilege to work with some expert and experienced professionals, which was truly an enriching experience.

My heartiest thanks to Dr. Abdelwahab Hamou-Lhadj, my supervisor, for showing confidence in me, providing me with initial spark for the topic, listening to my ideas no matter how vague they were and for constantly providing me support from his vast source of knowledge.

In addition, I would like to thank my lab mates and especially Mr. Amir Pirzadeh for their help. They took time out of their research to discuss with me about my topic and sometimes providing me useful tips and tricks to get the work done.

Lastly, I would like to thank my family in Iran. They always backed me up and had been supportive throughout the course of my studies.

Table of Contents

List of Tables

\

List of Figures

Chapter 1. Introduction

1.1. Problem and Motivation

Maintaining a large software system is not an easy task. Software engineers have often to understand what the system does before they can make any changes to the system. This is particularly important for those systems with poor documentation and for which the original designers have moved to other projects or companies.

In this thesis, we focus on techniques that permit the understanding of the behavioural aspects of software systems. These techniques often rely on tracing and run-time monitoring to generate information from a running system for further analysis. Traces, however, are overwhelmingly large, which hinder their proper analysis. Several trace abstraction have been proposed. Although these techniques vary in their design, they all focus on abstracting out the main content from the trace despite the trace being massive.

It is hard, however, to evaluate these techniques due to fact that they are often described in a non-formal way leaving room for ambiguity and misinterpretation. They also require extensive experimentations in order to validate their effectiveness. This appears to be due to the lack of a formal framework that can formally define the concept of traces and trace analysis methods.

The objective of this thesis is to present a formal framework for trace abstraction and correlation techniques. We believe that, if adopted, our framework can serve as the basis of many other techniques that can be defined and evaluated using the framework.

The framework focuses on trace abstraction techniques based on pattern detection [Hamou-Lhadj 03a, De Pauw 98], which are perhaps the most used trace simplification methods. We also present in this thesis a formalized trace correlation method that allow comparing traces generated from different systems. This is particularly important for comparing traces generated from two subsequent versions of the same system to understand the effort required to maintain it.

The framework is defined in a way that is easily extendible. One can add new definitions, methods, or any other analysis techniques.

The traces used in this thesis are traces of routine calls. We use the terms routine, method, procedure, and function interchangeably.

1.2. Research Contributions

The main contributions of this thesis are as follows:

- A complete formal framework that models traces of routine calls and related concepts. The framework also covers trace abstraction techniques based on pattern matching.
- A new approach for trace correlation where traces are compared based on their behavioural patterns. The approach is formalized.
- The application of the framework to several traces generated from different systems. The results show the effectiveness of our approach.
- The framework has been implemented in Java in the Eclipse environment.

1.3. Thesis Outline

The rest of the thesis is structured as follows:

- **Chapter 2:** This chapter starts with brief overview of related topics such as Program Comprehension, Software Evolution and Maintenance. The literature review includes trace abstraction techniques.
- **Chapter 3:** A formal framework is presented. The chapter starts with a formal representation of routine call traces. It continues with formalizing trace abstraction techniques based on pattern matching. After that, a new method for trace correlation is presented and formalized.
- **Chapter 4:** The evaluation of our approach is presented in this chapter. The chapter introduces the target systems with their different versions used in this thesis. The characteristics of traces generated from these systems are discussed. The results of applying the trace correlation process are presented and discussed.

- **Chapter 6:** We conclude the thesis in Chapter 6 with a summary of the main contributions, future works, and a concluding remark.

Chapter 2. Background

Software Engineering topics which are related to this thesis approach are namely, Software Evolution and Maintenance and Software Comprehension. These topics are described briefly according to the literatures and we will discuss the contribution of this thesis in these terms. A survey of routine call traces and pattern detection techniques is presented on which the unified framework of routine call traces is established. Edit distance of strings and tree-to-tree distance measures are also presented in this chapter.

2.1. Software Evolution and Maintenance

The major reason for software maintenance is the software aging. According to Parnas, software aging is inevitable [Parnas 94], but effective software maintenance can help slow down the process of aging. An important part of a software life cycle is maintenance [Lientz 80], is defined in IEEE standard 1219 as "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [IEEE 93].

The software evolution term does not have a standard definition but most of the time it is used as a substitute term for software maintenance. According to Bennett and Rajlich [Bennett 00], the software maintenance phase starts after the software development phase, i.e., after the software system is released. They further introduced the concept of a staged software lifecycle model in which development and maintenance were considered different phases in the software life cycle [Bennett 00].

Maintenance activities are classified into four categories [Lientz 80].

- Adaptive Maintenance: This type of maintenance includes user enhancements and modification to the existing software system to meet new user requirements.

- Perfective Maintenance: This involves making changes to the structure of the system in order to make it easier to extend, modify, and maintain.

- Corrective Maintenance: This type of maintenance deals with fixing software bugs in existing system functionality.

- Preventive Maintenance: This type of maintenance focuses on restructuring the existing system to prevent the system from bugs that may occur in the future.

Since finding the behavioral difference of a software version with previous versions and locating the part of the source code responsible for new behaviors is helpful for better understanding of new software, Trace correlation approach is useful in all types of maintenance.

2.2. Program Comprehension

According to Rugaber program comprehension refers to the process of acquiring knowledge about a program in order to increase the knowledge to be able to do such activities as bug correction, enhancement, reuse, and documentation [Rugaber 95]. Fjeldstad and Hamlen have stated that program comprehension accounts for 50% of the time spent on software maintenance activities [Fjeldstad 83]. Acquiring new knowledge of a system requires existing knowledge of a system [Mayrhauser 95] and trace

correlation approach can help to find out the parts of the system responsible for new behaviors by knowing the knowledge of the previous version behaviors. Newly acquired knowledge will become a part of system knowledge that is essential to support the understanding the system code. Based on their study, the authors conclude that software engineers possess two types of knowledge:

- General Knowledge: This type of knowledge is gained from past experience in the software engineering domain and is independent of the software under consideration.

- Software-Specific Knowledge: This knowledge represents their level of understanding of the software application under consideration.

In order to understand the system completely, software engineers use both general knowledge and software-specific knowledge of the system under consideration [Mayrhauser 95].

Documentation of systems are basically representing high-level view of the systems while implementation contains more low-level programming details, and this is why program comprehension is not an easy job [Rugaber 95]. If maintainers know behaviours of previous versions of a system, the correlation approach may help them to target part of the implementation responsible for new behaviours. Our approach can help maintainers to find the proportion of new behaviours in new systems.

2.3. Static and Dynamic Analysis

There exist two main techniques for analysis software systems: static and dynamic analysis techniques.

Static analysis is usually based on the source code. The objective is to extract high-level views from low-level implementation. The resulting views often contain the static components of the system and the way they interact with each other.

Dynamic analysis, the focus of this thesis, is concerned with running the system and analyzing the way it behaves. It is a suitable analysis technique if one wants to understand how a particular aspect of the system is implemented instead of the whole system as it is the case for static analysis. Dynamic analysis also allows relating program input to output which can help maintainers understand the way the system behaves.

Run-time information is typically represented in the form of execution traces. Traces, however, have been difficult to work with since they are often long. There are various ways for generating traces including source code instrumentation (add probes – printout statements – to the system). The source code is then recompiled and executed. If the source code is not available, once can also instrument the operating system or the virtual machine.

2.4. Trace Abstraction

To reduce the size of traces, several trace abstraction techniques have been proposed. The objective is to help maintainers understand the main content conveyed in a trace instead of spending time and effort browsing a large trace, which is a difficult task even with tool support. Trace abstraction techniques can be divided into four categories:

- Pattern detection
- Sampling
- Grouping
- Visualization

2.4.1. Pattern detection

A trace pattern is defined as a sequence of calls that is repeated non-contiguously in the trace [De Pauw 98, Hamou-Lhadj 03a]. The idea is that once detected, software engineers will only need to look at them once, which should reduce the effort required to go through the trace. Trace patterns are also believed to encapsulate important behaviour invoked in a trace, for example, a computation that is repeated several times in the trace. Therefore, understanding trace patterns can help understand the main thing that happens in the trace.

Patterns, however, are only effective if generalized. Using identical matching among sequences of calls will lead to several patterns that might differ only slightly. Several matching criteria such as 'ignoring order of calls' and 'ignoring number of repetitions' have been proposed in the literature [De Pauw 98, Hamou-Lhadj 03a]. In this thesis, we focus on modeling trace patterns and the various matching criteria associate with them.

2.4.2. Sampling

Sampling is a trace abstraction technique in which the trace elements are sampled according to sampling distance [Chan 03, Whaley 00, Dugerdil 07]. A sampling distance

could be automatically determined by knowing other parameters such as the original size of the trace and the final expected size of the sampled trace. If the sampling distance is *n* it means that every *n* events in the trace should be considered. Using sampling, we can obtain a trace which is *n* times smaller than the original trace.

Sampling, however, has been shown to be limited in many ways. The problem is that it is very hard to define sampling parameters that can result in a sampled trace that is reflective of the original trace. In addition, sampling parameters of even the same system might not work for all the scenarios.

2.4.3. Grouping

In [Kuhn 06] a monotone subsequence summarization technique has been introduced where a trace is composed of monotone subsequences separated by pointwise discontinuities. A pointwise discontinuity occurs when the nesting level suddenly drops as execution continues with the latest sibling of the previous events. Pointwise discontinuities could be used as delimiters between groups of routines. A gap size constant could be set to find the discontinuity pointwises.

2.4.4. Visualization

Several tools have been proposed to help software engineers work efficiently with large traces (e.g. [Hamou-Lhadj 04], [De pauw 93]). These tools rely on visualization schemes

that range from simple features such as zooming, highlighting, searching to advanced layout including 3D layout and animations techniques.

These techniques, however, are tightly coupled to the visualization method that is used which hinders their reuse. In addition, they require a lot of intervention from the users, which if often a difficult task.

2.5. The Edit Distance

One of the key concepts used in this thesis is a way to compare traces. For this, we rely on using the edit distance between trees. Trees are one of the most useful data structures in many aspects of the science such as compiler design, information retrieval, graph transformation pattern recognition, image processing, chemistry and etc.

The tree pattern matching problem to compare trees is related to the problem of string pattern matching. Several distance measures for comparing traces have been proposed in [Tai 79, Jiang 95, Selkow 77, Tanaka 88, Valiente 01(a), Yang 91, Valiente 01(b)], which essentially differ on the underlying notion of subtrees. Distance measure between trees is the generalization of the edit distance between strings [Gusfield 97, Stephen 94].

The edit distance between two trees refers to the cost of transforming one tree to the other tree. That is, the distance between two trees is given by the shortest or the least expensive, in terms of operations, sequence of elementary edit operations (insertion, substitution, and deletion of labeled nodes) that allow transforming one tree into the other. Despite their original definition in terms of elementary edit operations, distance measures between trees can also be stated in terms of mappings that is substituting similar or isomorphic subtrees to each other.

Chapter 3. Framework for Trace Analysis

In general providing a framework of the problem domain may simplify our work, especially when we are dealing with big definitions, concepts, algorithms and calculations. In this chapter we present a unified framework in which most of the routine call traces' concepts and most of the pattern detection definitions and algorithms are defined in a formal way. However every new research might define its own framework, a unified and general framework provides the opportunity of comparing different techniques. The framework could be extended to contain other abstraction techniques and not only pattern detection.

The framework should be simple and complete. The purpose of framework is to simplify the calculations and definitions, all concepts should be understandable and easy to use, so the simplicity may motivate everyone to use the framework. When we say the framework should be complete it basically refers to the accuracy of the framework in which all definitions are accurate and related to each other logically and there is no contradiction. If framework contains all concepts or at least all primary concepts it could provide the basements for the users to define new concepts. Finally a simple and complete framework could be a good help to invent a new technique and compare it with previous techniques.

3.1. Definition of Trace

A trace of routine calls is a tree structure where the root represents the first call, followed by subsequent calls made to different routines. We represent each node of a call tree as a triple (label, nesting level, position) where:

- The label refers to the full name of the routine being invoked. The full name usually includes the name of the routine itself preceded with any other information that can uniquely identify the routine. For example, in an object-oriented system the full name could include the class where the routine is defined as well as the package that defines the class.

- The nesting level represents the nesting relationship among calls. By convention, we assign to the root call a nesting level 0, the routines called by the root have a nesting level 1, etc.

- The position, in our formalization model, represents the unique location of the node in the trace where the routine is invoked. We will use this position to define more operations on traces such as formalizing paths from one node to another.

Figure 3.1 shows an example of a routine call trace. In this example, the root node is represented as (r_1, 0, 0), and (r_2, 1, 1) is the first callee of the root so the position is incremented by one because it is the first call after the root, the nesting level is one level upper than its parent which is the root. We can see that (r_4, 2, 3) and (r_4, 2, 7) both have the same label but they do not have the same position. The nesting level of the direct children is always one level higher than the nesting level of their parent as shown in Figure 3.1.

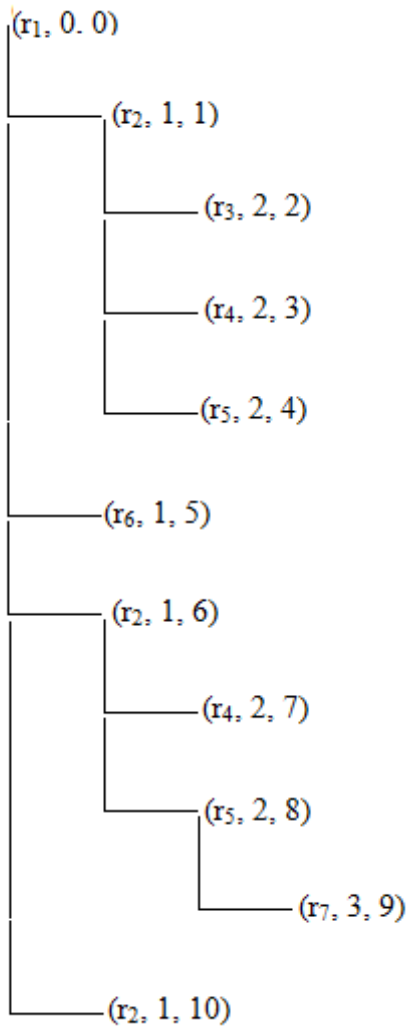

Figure 3-1 : Example of a trace of routine calls

Definition 3.1: We define a set T as the set of all nodes of the trace (i.e., the set of routine calls). If *u* is an element of the set T, then we use the notation *u.l* to mean the label of the node *u*, *u.n* the nesting level, and *u.p* the position of the call in the trace. More formally:

$$T = \{(l, n, p) \mid n \geq 0 \wedge p \geq 0 \wedge (l \text{ is string})\} \qquad (3.1)$$

Furthermore, if a subtree rooted at a node *u* satisfies the conditions of a trace (i.e. root has a nesting level 0, thewe say that this subtree is a proper trace that means the subtree rooted at *u* is an ordered tree and we show it with:

$$isTrace(u) \Leftrightarrow \text{subtree rooted at node u is a trace.} \quad (3.2)$$

Definition 3.2: We use the notation T[i] to mean the node at position `i` of the tree T. More formally:

$$T[i] = u \Leftrightarrow u \in T \wedge u.p = i \quad (3.3)$$

Definition 3.3: Each node except the root has a unique parent with a nesting level decremented. The position of the parent is less than the position of the callees because the parent must be called first - and this case it occurs in the trace - before the routines that it calls. This is represented in Equation 3.4.

$$parent(u) = v : u \neq root \Leftrightarrow \forall i : i.n = u.n - 1 \wedge i.p < u.p \Rightarrow v.p \geq i.p \quad (3.4)$$

It means that if *v* is the direct parent of *u* then the position of *v* is the maximum position among all nodes whose nesting level is one level less than the nesting level of *u*. In Figure 3.1 we can see that for the node $(r_5, 2, 8)$ the parent node is (r2, 1, 6) since this is the node that has the maximum position of the nodes whose nesting level is less than the nesting level of $(r_5, 2, 8)$.

Definition 3.4: The set of callees of a node consists of all the nodes for which c is parent as shown in Equation 3.5.

$$callee(u) = \{c \mid parent(c) = u\} \quad (3.5)$$

It should be noted that the function *callee* will return the direct children of a given node and not all its descendants.

Definition 3.5: We define the path from the root to a particular node using the below function, which is a recursive function that starts from node and collects the parents nodes until the root is reached.

$$path(c) = \begin{cases} \emptyset & \text{if } c \text{ is the root} \\ parent(c) \cup path(parent(c)) & \text{else} \end{cases} \quad (3.6)$$

The path returns a set that contains all the nodes who are the ancestor of the target node. For example, in Figure 3.1, the path for the node $(r_4, 2, 7)$ is the set $\{(r_2, 1, 6), (r_1, 0, 0)\}$.

Definition 3.6: A subtree can be defined using the notion of path as follows:

$$subTree(u) = \{c \mid u \in path(c)\} \quad (3.7)$$

3.2. Formalization of Pattern Based Trace Abstraction Techniques

As discussed in the background section, traces tend to be overwhelmingly large. Several trace abstraction techniques have been proposed. Although these techniques vary in their design their objective is to reduce the information contained in a trace while keeping the main content.

Among these techniques, the most popular ones are the ones based on detecting recurrent patterns. The idea is that a pattern in a trace could indicate that something important in the trace is happening. Software engineers can focus on these patterns when trying to understand the content of a large trace. Patterns have been showed to be useful in several software maintenance tasks such as understanding where a fault occurred, adding a new feature, etc.

In this section, we formalize the various concepts related to trace abstraction based on pattern detection. We particularly focus on the techniques that permit matching sequences of calls based on various matching criteria. These criteria are important since detecting identical patterns might not be useful because of the large number of patterns that may exist in a trace [De Pauw 98, Hamou-Lhadj 02].

3.2.1. Definition of a Trace Pattern

Definition 3.7: We define a trace pattern as any sequence of calls that are repeated non-contiguously more than once in a trace. It should be noted here that the focus on non-contiguous repetitions since contiguous repetitions are due to loops and recursion and are often removed from the trace and replaced with one sequence and the number of contiguous repetitions of this sequence.

The best way to represent patterns in a routine call trace is by turning the trace into an ordered directed graph (DAG) as shown in Figure 3.3. This is because any rooted tree can be turned into its most compact form which is represented as an ordered DAG by

representing repetitions only once [Hamou-Lhadj 03a]. In this figure, we can see that the subtree rooted at node C is represented only once in the DAG that corresponds to the tree.

Hamou-Lhadj et al. presented an efficient algorithm for turning a tree into an ordered DAG [Hamou-Lhadj 03a]. They also introduced the concept of comprehension units which represents the nodes of the DAG generated from transforming a call tree. According to them, the understanding of trace content is often reduced of understanding its comprehension units since software engineers only need to understand sequence once and reuse this understanding whenever the sequence appears again in the trace. In this thesis, we also use the term comprehension unit in our formalization framework.

Definition 3.8: We define the symbol $comp_i$ to refer a comprehension unit of a trace. Every *comp* represents an equivalence class such that two nodes *u* and *v* of a tree are part of this equivalent class if and only if the subtrees rooted at these nodes are similar. Similarity is measured in various ways as we will describe in Section 3.2.2. In the absence of similarity measures, the two subtrees must be identical - They have the same structure and the nodes at corresponding places have identical labels. The set of all distinct subtrees of a given tree consists of a subtree from each equivalence class.

$$u, v \in comp_x \Rightarrow subtree(u) \text{ and } subtree(v) \text{ are similar} \qquad (3.8)$$

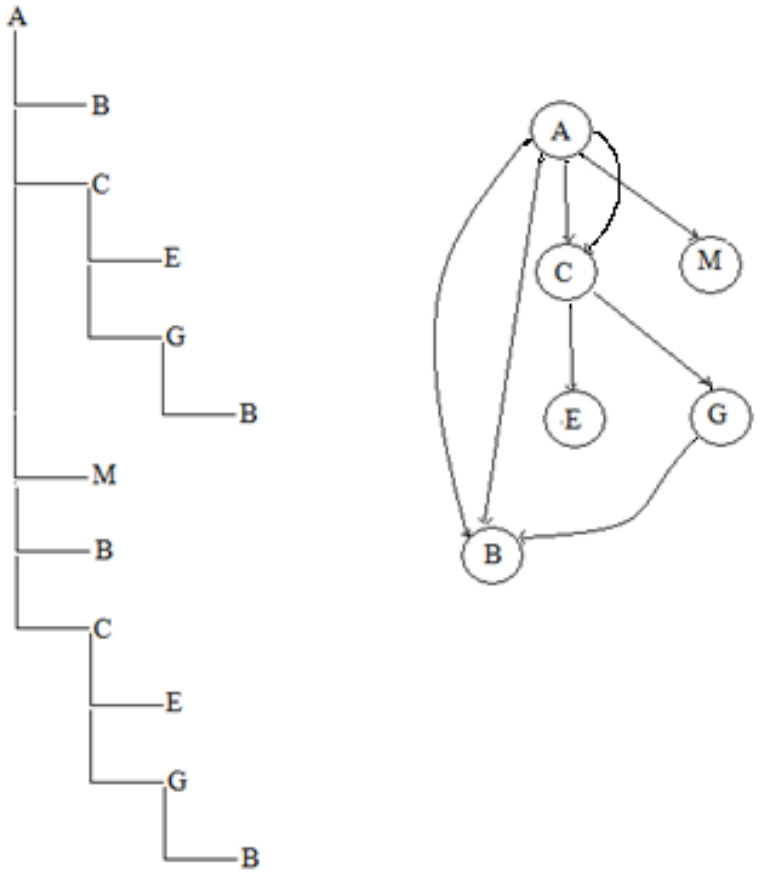

Figure 3-2 :Example of a trace of routine calls and its DAG graph

Figure 3.2 shows the comprehension units of the tree T, which are represented by the following sets:

$A \in PClass_0$

$B \in PClass_1$

$C \in PClass_2$

$E \in PClass_3$

$G \in PClass_4$

$M \in PClass_5$

$PClass_1$, $PClass_2$, $PClass_3$ and $PClass_4$ are representing 4 class of patterns.

3.2.2. Matching Criteria Predicates

Identical matching when comparing subtrees will lead to many patterns that might only differ slightly. Several criteria have been proposed in which similar (not necessarily identical) subtrees could be deemed as instances of the same pattern [Hamou-Lhadj03(b)]. In this section we have developed predicates that describe the most cited matching criteria. The predicates take two nodes of a trace and return whether the subtrees rooted at these nodes are similar based on the specified criteria or not.

Definition 3.9 - Identity: Two subtrees are identical if their roots have the same label and both subtrees have the same structure and the same children. Figure 3.3 shows an example of identical subtrees. In this figure, the subtrees rooted at B are identical. Based on this matching criterion, the corresponding DAG (as shown in Figure 3.2) treats this two subtrees as sequences of the same pattern.

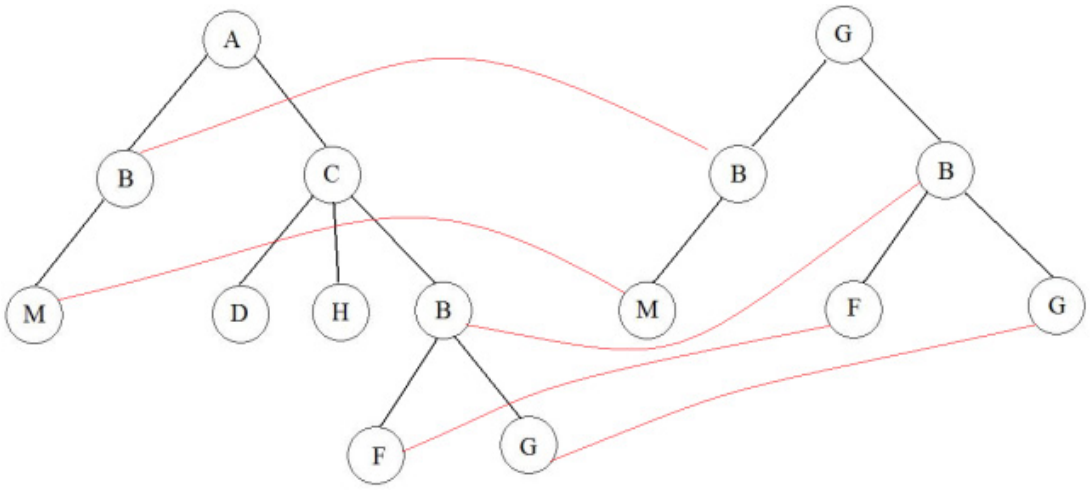

Figure 3-3: Identical subtrees are mapped to each other

The below *fIdentical* function has been developed to formalize the identity matching criterion.

$$fIdentical(c_1,c_2) \Leftrightarrow [(c_1.l=c_2.l) \wedge callees(c_1)=callees(c_2)=\emptyset)] \vee [\forall c \in callees(c_1) : \exists C' \in callees(c_2) : (c.p - c_1.p = c'.p - c_2.p) \wedge fIdentical(c,c')]$$

Moreover

$$fIdentical(c_1, c_2) \Leftrightarrow fIdentical(c_2, c_1) \quad (3.9)$$

The *fIdentical()* function is defined recursively. The function stops when it meets two nodes c_1 and c_2 which have the same label and are both leaves. Otherwise if for every callee of c_1 there is a callee in c_1 at the same position ($c.p - c_1.p = c'.p - c_2.p$) and that the two callees are identical so two calls c_1 and c_2 are identically similar.

Definition 3.10 - Class Identity: In object-oriented programming, usually the label of each node consists of the name of the package, the class that defines the method, the object on which the method is invoked, and the method name. We can vary the

information found in the label to group sequences of calls. For example, if we ignopred the object and the method name and decide to consider only the class and package name then we will end up with more sequences of calls that can be deemed as instances of the same pattern. This is sometimes useful when we do not need to look at every single object and method being invoked and we only need to understand the main interactiosn that occur between classes. Similarly, one can focus on the interactions between packages. In Equation 3.10, we show how similarity based on class name can be formalized. Similar equations can be developed depending on which element of the call label is considered. We use the notation of *c.l.className* to compare the labels based on class names only.

$$fClassIdentical(c_1,c_2) \Leftrightarrow [(c_1.l.className=c_2.l.className) \wedge callees(c_1) = callees(c_2)=\emptyset)] \vee [\forall c \in callees(c_1) : \exists C' \in callees(c_2) : (c.p - c_1.p = c'.p - c_2.p) \wedge fClassIdentical(c,c')]$$

Moreover

$$fClassIdentical(c_1, c_2) \Leftrightarrow fClassIdentical(c_2, c_1) \quad (3.10)$$

Definition 3.11 – Ignore Order of Calls: In many software maintenance tasks, there is no need to understand every single call invoked in a trace or even the order of calls. The 'ignore order' matching criteria leads to greater generalization of patterns by ignoring the order of calls when matching two subsequences of calls. Figure 3.4 shows an example of

six similar subtrees based on the ignore order criterion. The two subtrees rooted at node *C* can be considered similar by ignoring the order of invocation of their children.

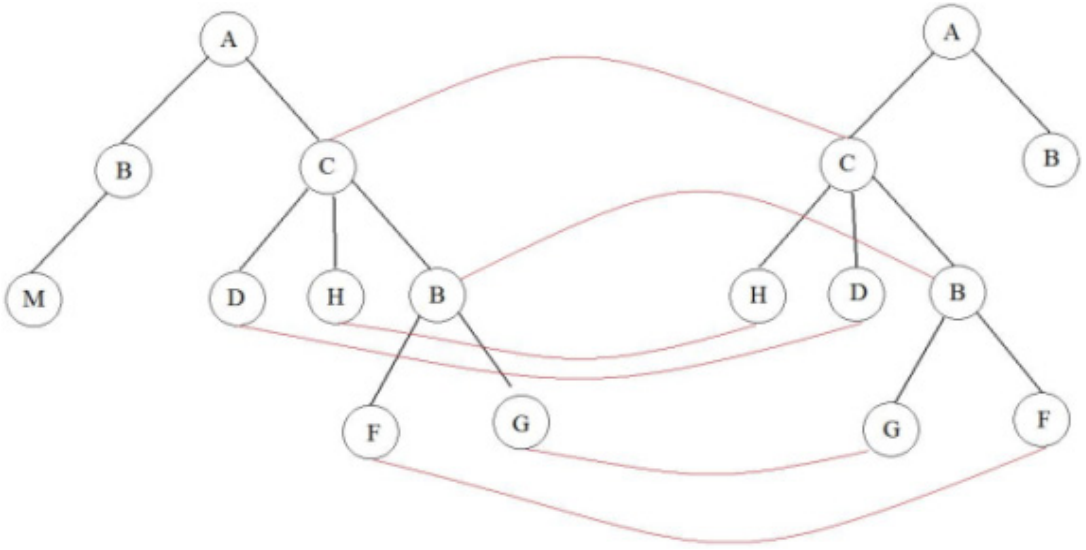

Figure 3-4 : Similar subtrees are mapped to each other under ignore order criteria

The below function formalizes the 'ignore order' matching criterion. The function is recursive one and the stopping condition is when two subtrees have the same name and appear at the leaf level ($[(c_1.l=c_2.l) \wedge (callees(c_1)=callees(c_2)=\emptyset)]$). The second term of the disjunction provides the fact that the number of similar children of the first call and the second call should be the same. We can see that in this criterion the number of the similar children of two sequences of calls should be the same but their order is not important, so we use the cardinality symbol " | | " to model this.

$$fIgnoreOrder(c_1,c_2) \Leftrightarrow [(c_1.l=c_2.l) \wedge (callees(c_1)=callees(c_2)=\emptyset)] \vee |\{c'|c' \in callees(c_1) \wedge \exists c'' \in callees(c_2) : fIgnoreOrder(c',c'')\}| = |\{c'|c' \in Callees(C_2) \wedge \exists c'' \in callees(c_1) : fIgnoreOrder(c',c'')\}|$$

Moreover

$$fIgnoreOrder(c_1,c_2) \Leftrightarrow fIgnoreOrder(c_2,c_1) \qquad (3.11)$$

Definition 3.12 - The Set Criterion: In this criterion, we are treating all children of a subtree as a set (the order and the number of repetitions are ignored). If two sets are exactly the same, two corresponding subtrees are considered similar. The set criterion should lead to a greater compaction of the trace since many sequences of calls can now be considered as instances of the same pattern. The resulting ordered DAG after applying the set criterion is often very compact as shown by Hamou-Lhadj et al. in [Hamou-Lhadj 03a].

Figure 3.5 shows an example of applying the set criterion. The two subtrees rooted at *C* are considered similar by treating their calls as sets (i.e. ignoring the order of calls and the number of repetitions).

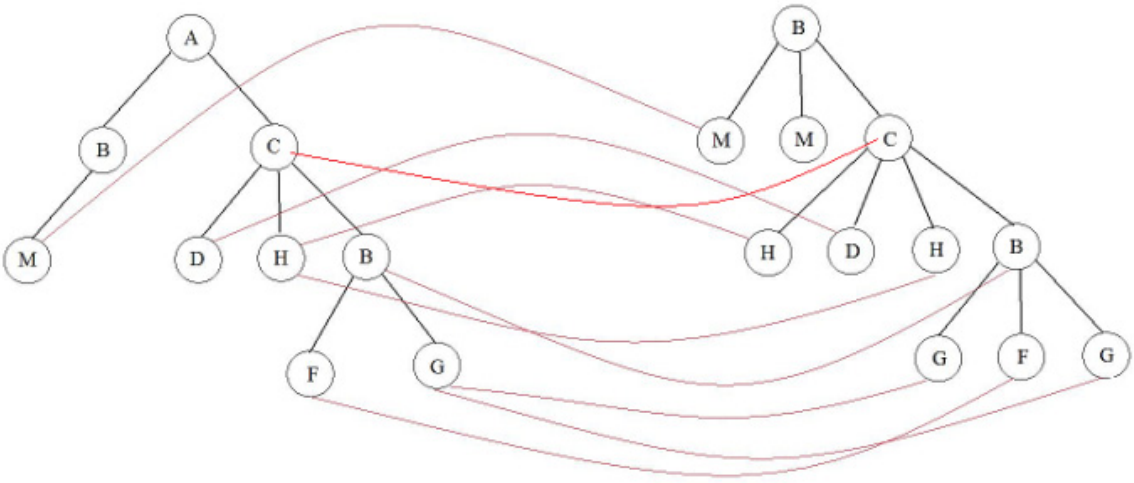

Figure 3-5 : similar subtrees are mapped to each other under set criteria

The *fSet* function of Equation 3.12 formalizes the set criterion. It is again a recursive function. The sopping condition for the function *fSet()* is when two nodes are leaves and similar. Otherwise we need to make sure that for every callee of the first call there is at least one similar callee among the children of the second call ($\forall c \in callees(c_1) : \exists c' \in callees(c_2) : fSet(c,c')$).

$$fSet(c_1,c_2) \Leftrightarrow [(c_1.l=c_2.l) \wedge callees(c_1)=callees(c_2)=\emptyset)] \vee [\forall c \in callees(c_1) : \exists c' \in callees(c_2) : fSet(c,c')]$$

Moreover

$$fSet(c_1,c_2) \Leftrightarrow fSet(c_2,c_1) \qquad (3.12)$$

Definition 3.13 - Depth Limiting: Two subtrees can be considered similar if they are compared up to a certain depth and the rest of the calls that are beyond this depth are ignored. Consider the following example:

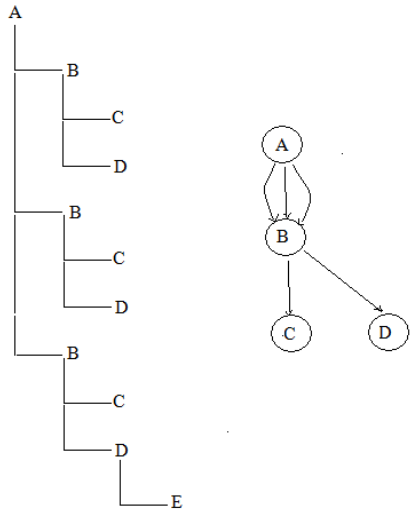

Figure 3-6 : An execution trace compared to a certain depth

In Figure 3.6, if we limit the depth of the tree to 2 (the root has depth 0), then the node labeled E is ignored (since it appears at depth 3). In this case, the whole trace can be reduced to A calling B which calls C and D. Since the sequence B calling C and D is repeated contiguously then a significant reduction is obtained as shown in the corresponding DAG. Usually the depth limiting criterion is combined with other criterion for example the identity matching criterion as in this example.

The following function shows how depth_limiting combined with the identity matching criterion is formalized. The only condition that we need to add to the identity criterion function is "$c_1.n \leq d \wedge c_2.n \leq d$ " that checks the nesting level. Other definitions can derive from this definition by varying the matching criterion that is used when the depth limiting is used.

$$fIdentical\_depthLimit(c_1,c_2,d) \Leftrightarrow [((c_1.l=c_2.l) \wedge callees(c_1) = callees(c_2)=\emptyset) \wedge c_1.n \leq d \wedge c_2.n \leq d) \vee ((c_1.l=c_2.l) \wedge c_1.n = d \wedge c_2.n = d)] \vee [\forall c \in callees(c_1) : \exists C' \in callees(c_2) : (c.p - c_1.p = c'.p - c_2.p) \wedge fIdentical\_depthLimit(c,c', d)]$$

Moreover

$$fIdentical\_depthLimit(c_1, c_2, d) \Leftrightarrow fIdentical\_depthLimit(c_2, c_1, d) \quad (3.13)$$

Definition 3.14 - Ignoring Utilities: Hamou-Lhadj et al. have introduced a new matching criterion known as the removal of utilities [Hamou-Lhadj 06]. According to them utilities are any component that implements low-level functions that is not needed to understand the overall behavior invoked in a trace. They argued that these utilities should be removed when looking for trace patterns. The removal utilities permit better generalization of trace patterns without necessarily losing important information.

In Figure 3.7, the two sequences rooted at B can be considered similar if u1 and u2 (utilities) are ignored during the matching process. It should be noted, however, that it is not always simple to identify what constitutes a utility. For this purpose, several techniques have been proposed including a utilityhood metric that was proposed by Hamou-Lhadj et al. [Hamou-Lhadj 06] which measures the extent to which a system component could be considered as a utility. The detection of utilities is beyond the scope of this thesis.

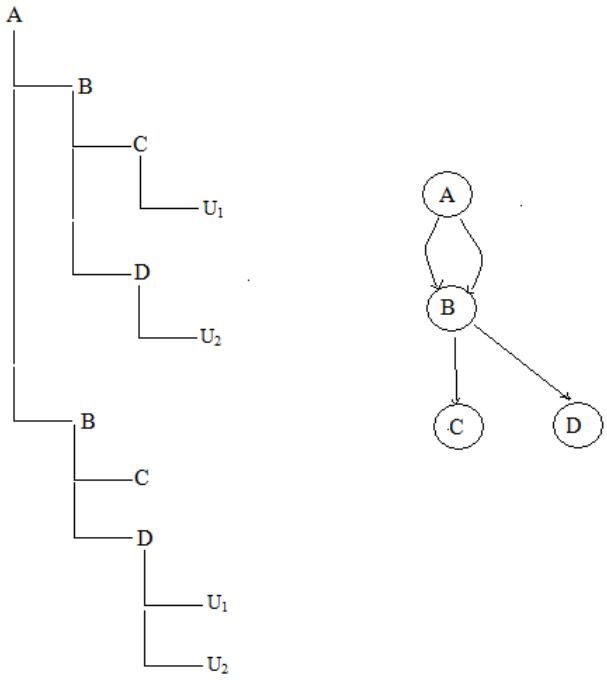

Figure 3-7 : Two sequences of calls containing utility methods

To apply utility removal criterion we need to remove all utilities first from the set *T* and then we can use one of the other criterion to abstract the trace.

Definition 3.15 - Edit Distance: The edit distance is a measure of similarity between two subtrees [Tai 79]. More particularly, it refers to the number of edit operations (insertion,

substitution, deletion) that are required to transform one subtree to another subtree. If the subtrees are isomorphic then the number of operations is zero. The usage of the edit distance for pattern matching ha first been proposed by Hamou-Lhadj et al. in [Hamou-Lhadj 03a]. The authors argued that there are many situations where two subtrees might slightly vary and that this variation cannot be captured with existing matching criteria such the ignore number of repetitions, ignore order, etc., which tend to be more structural. In such case, one can measure the difference between two subtrees and decide based on a threshold whether the subtrees can be considered as instances of the same pattern or not.

The application of the edit distance requires a threshold to be given as input. For example in Figure 3.8, if the minimum edit distance is four then two subtree rooted at node A can be considered similar because when the transformation of the first subtree into the second subtrees rooted at A required to delete the nodes E and F and insert the node G that is three edit operations, so the distance between the two subtrees is three which is less than four.

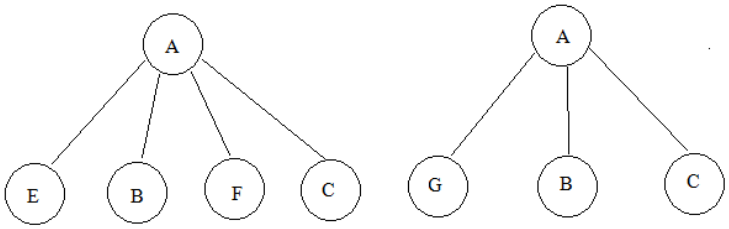

Figure 3-8 : Two sequences of calls with distance of three

The formal definition of the edit distance is deferred to the Section 3.3, where a trace correlation mechanism is presented in which the edit distance is the main mechanism.

Definition 3.16 – Flattening: Flattening consists of ignoring the hierarchal structure of the calls and look at the nodes as they are flattened linearly and compare them. This matching criterion was proposed by De Pauw et al. [De Pauw 98]. The authors argued that it can lead to better generalization, especially in situation where the main objective is to simply understand the key components being invoked and not how they are structure.

In Figure 3.9, two subtrees rooted at node C can be considered similar if their hieratical structure is ignored, we can see that in the first subtree, the node B occurs twice but once flattened, the repetition is also ignored.

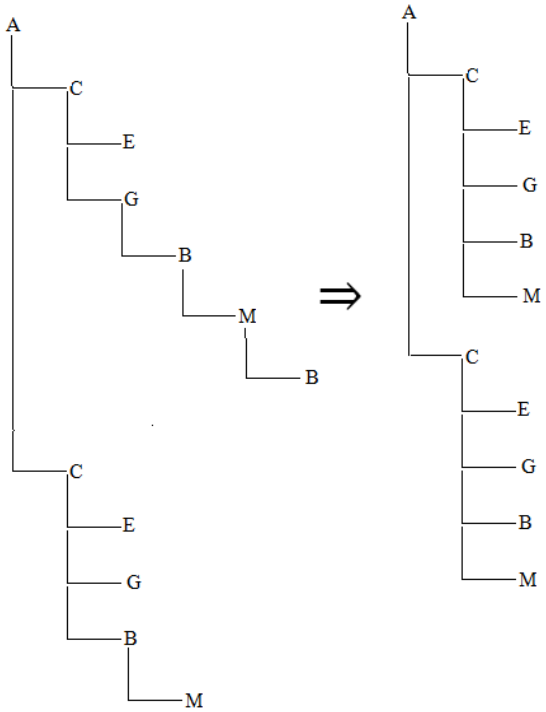

Figure 3-9 : Applying the flattening matching criterion

In order to provide a formal definition for the flattening criterion we define a new subtree function which is *subTree_label()* that behaves exactly like the subtree function defined

in Equation 3.6 except that it returns a set of calls in terms of their label and not as triples (label, nesting level, position).

The formal definition for the predicate *fFlattening()* is provided below.

$$fFtattening(c_1, c_2) \Leftrightarrow c_1.l = c_2.l \wedge subTree\_label(c_1) = subTree\_label(c_2) \quad (3.14)$$

The expression *"subTree_label(c_1) = subTree_label(c_2)"* is used to mean that two sets returned by *subTree_label* must be the same.

3.3. Formalization of Trace Correlation

3.3.1. Introduction

Finding the correlation between two traces of routine calls may help in a variety of software maintenance tasks including:

- Understanding how specific features of subsequent versions of the same system differ. This is particularly important to estimate the amount of effort required to understand the changes made to an existing system, which in turn can help maintain and test the new system.

- Checking the consistency of the running system with design specification. This can help determine if the system does what is supposed to do. We recognize, however, that this require more than trace correlation since high-level design specifications do not contain detailed information that would exist in a trace. In such a case, trace

abstraction (based on the techniques presented in the previous section) should be employed before the correlation process takes place.

- Trace correlation has also been used in the area of security, especially in the context of redundancy and diversity architectures. Two instances of the same system run on different nodes. One node is kept offline and is considered secure whereas the other node is in operation and therefore vulnerable to attacks. The execution traces generated from both instances are correlated to detect any deviations from normalcy, which could indicate the presence of an attack.

In this thesis, we propose using the edit distance to measure the correlation between two traces. However, comparing traces based on their mere events might turn to be inefficient due to the considerably large number of events generated. Instead, we propose, in this thesis, to compare traces based on their behavioral patterns. As discussed in the previous section, trace patterns are believed to represent the main behavior invoked in a trace. Two traces should therefore be similar if they exhibit the same behavior despite the number of events they contain. In addition, to allow enough generalization, we do not limit ourselves to identical patterns. Instead, we propose using different matching criteria to provide enough flexibility to users to vary the correlation criteria. For example, users might want to ignore the number of repetitions when comparing two patterns of different traces.

3.3.2. The Edit Distance

As mentioned in the previous section, the edit distance between two trees refers to the cost of transforming one tree into the other one [Tai 79]. In other words, the distance between

two trees is given by the shortest or the least-costly sequence of elementary edit operations (insertion, substitution, and deletion of nodes) that allow transforming one tree into the other. Despite their original definition in terms of elementary edit operations, distance measures between trees can also be stated in terms of mappings that is substituting similar or isomorphic subtrees to each other.

One of the most efficient way to find the distance between two trees T_1 and T_2 is first to find the largest similar (or isomorphic) structures of two trees, where similar structures identify the unchanged part of the T_1 during transformation of T_1 to T_2.

Definition 3.15 - Mapping Set*:* The mapping set represents a one-to-one correspondence between the nodes of T_1 to T_2 when the ancestors' order is preserved. We mean by the ancestor order needs to be preserved that if two nodes are mapped to each other we cannot map the parent of the first node to a child of the second node. In other words, the parents need to be mapped to each other and the children need to be mapped to each other.

The formula 3.15 models formally the concept of mapping set. The pair *(i, j)* in *M* establishes a substitution of $T_1[i]$ by $T_2[j]$.

$$M = \{ (i, j) \mid T_1[i] \in T_1 \wedge T_2[j] \in T_2 \} \qquad (3.15)$$

We also define two other sets *I* and *J*. The set *I* refers to the all nodes of T_1 for which we could not find a correspondence in T_2 and the nodes in the set *J* are the T_2's nodes that are not included in the mapping set (i.e., they do not have corresponding nodes in T_1).

$$I = \{ T_1[i] \mid \nexists j : (i, j) \in M \} \qquad (3.16)$$

$$J = \{ T_2[j] \mid \nexists i : (i, j) \in M \} \qquad (3.17)$$

In order to transform T_1 to T_2 we need to substitute the nodes of T_1 which are in the mapping set by the nodes of T_2 which are in the mapping set and also remove the nodes in I from T_1 and insert the nodes of J in to T_1. The distance between T_1 and T_2 is represented by the following function:

$$dist(T_1, T_2) = |M|p + |I|q + |J|r \qquad (3.18)$$

where p is the cost of substitution, q is the cost of deletion, and r is the cost of insertion.

We can easily see that the edit distance is strongly related to the size of the mapping set. Given that there are many ways to transform T_1 into T_2 (resulting in several different mapping sets), we are concerned with finding a mapping that can reduce the cost of the edit distance. In general, a large mapping set between T_1 and T_2 represents that less modification effort (i.e., deletion and insertion) needs to be made to transform T_1 to T_2. This cannot be done unless we find the largest similar forests between the two trees T_1 and T_2. There are also a number of conditions that need to be satisfied when finding the distance between the ordered trees T_1 and T_2 [Tai 79]:

If (i_1, j_1), (i_2, j_2) and $(i, j) \in M$ then (3.19)

a) $1 \le i \le |T_1| \ \wedge 1 \le j \le |T_2|$

b) $i_1 = i_2 \Leftrightarrow j_1 = j_2$

c) $i_1 < i_2 \Leftrightarrow j_1 < j_2$

d) $T_1[i_1]$ is an ancestor (descendant) of $T_1[i_2] \Leftrightarrow T_2[j_1]$ is an ancestor (descendant) of $T_2[j_2]$

3.3.3 Pattern-Based Trace Correlation

As we mentioned earlier, we use the concept of patterns to compare two traces instead of relying of low-level events invoked in a trace. A one-to-one correspondence between the roots of the similar patterns (subtrees) in the two trees could generate a mapping set in which the pairs refer to the similar patterns (more precisely, they refer to the roots of the patterns). Matching criteria are used to match patterns to each other so as to avoid mere identical matching which can cause some slightly different patterns to be unmapped. Moreover, since we need to find the largest common forest, we can map every subtree which occurs at least once in each tree. We can also map the leaves to the ones in the other tree. Using this kind of mapping we can find a distance between the two trees.

However, this requires refining the mapping set definition presented in formula 3.17 to consider the matching criteria used to map subtrees from each trace to each other. In the following paragraph, we show how the identity, ignore order, and treating the calls as a set are defined in the framework. It should be noted that:

a) The ignore repetitions matching criterion is not defined. Instead, we propose removing contiguous repetitions before processing the traces. This preprocessing stage eliminates the necessity to use the 'ignore number of repetitions' criterion.

b) Not all the matching criteria are taken into account in this thesis. However, we anticipate the other matching criteria can readily be defined based on the framework presented here.

The Identity Matching Criterion:

Using this matching criterion, two subtrees from the traces are matched if they are isomorphic (same structure and children). In other words:

If (i , j), (i_1 , j_1), (i_2 , j_2), $(u , v) \in M$ (mapping set) then the following conditions must hold:

a) *i and j have the same number of children*

b) $i.l = j.l$

c) $i_1 = j_1 \Leftrightarrow i_2 = j_2$

d) $(u , v) \in M \Rightarrow \forall i \in children(u) \Rightarrow \exists j \in children(v) : (i.p - u.p = j.p - v.p) \wedge (i , j) \in M$

Figure 3.10 shows an example of a pattern-based mapping scheme using the identity matching criterion.

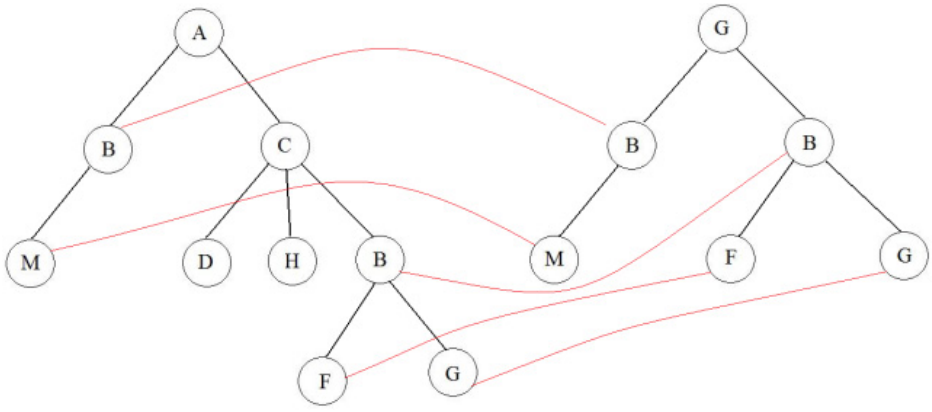

Figure 3-10 : The subtrees of the two traces are mapped using the 'identity' matching criterion

The Ignore Order Matching Criterion:

Using this matching criterion, two subtrees from the two traces are matched if they contain the same children no matter in which order they appear. In other words:

If $(i, j), (i_1, j_1), (i_2, j_2), (u, v) \in M$ (mapping set) then the following conditions must hold:

a) *i and j have the same number of children*

b) $i.l = j.l$

c) $i_1 = j_1 \Leftrightarrow i_2 = j_2$

d) $(u, v) \in M \Rightarrow \forall i \in children(u) \Rightarrow \exists j \in children(v) : (i, j) \in M$

An example of how subtrees from two trace are mapped using the 'ignore order' criterion is shown in Figure 3.11. In this figure, the subtrees rooted at C and B are mapped to eeach other despite the differences in the order of calls.

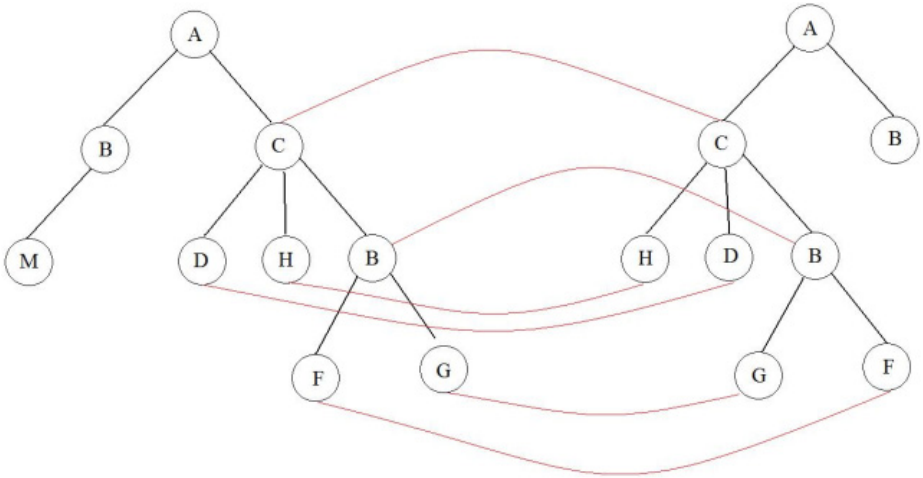

Figure 3-11 : Mapping subtrees using the 'ignore order' criterion

The Set Matching Criterion:

The set matching criterion enables the mapping between two subtrees by treating their children as a set. In other words, the order of calls and repetitions of the same calls are ignored. More formally, if $(i, j), (i_1, j_1), (i_2, j_2), (u, v) \in M$ (mapping set) then the following conditions must hold:

a) $i.l = j.l$

b) $(u, v) \in M \Rightarrow \forall i \in children(u) \Rightarrow \exists j \in children(v) : (i, j) \in M$

c) $(u, v) \in M \Rightarrow \forall j \in children(v) \Rightarrow \exists i \in children(u) : (i, j) \in M$

Figure 3.12 shows an example on how the set matching criterion is used. In this figure, the subtrees rooted at C, for example, are considered similar by treating their children as a set.

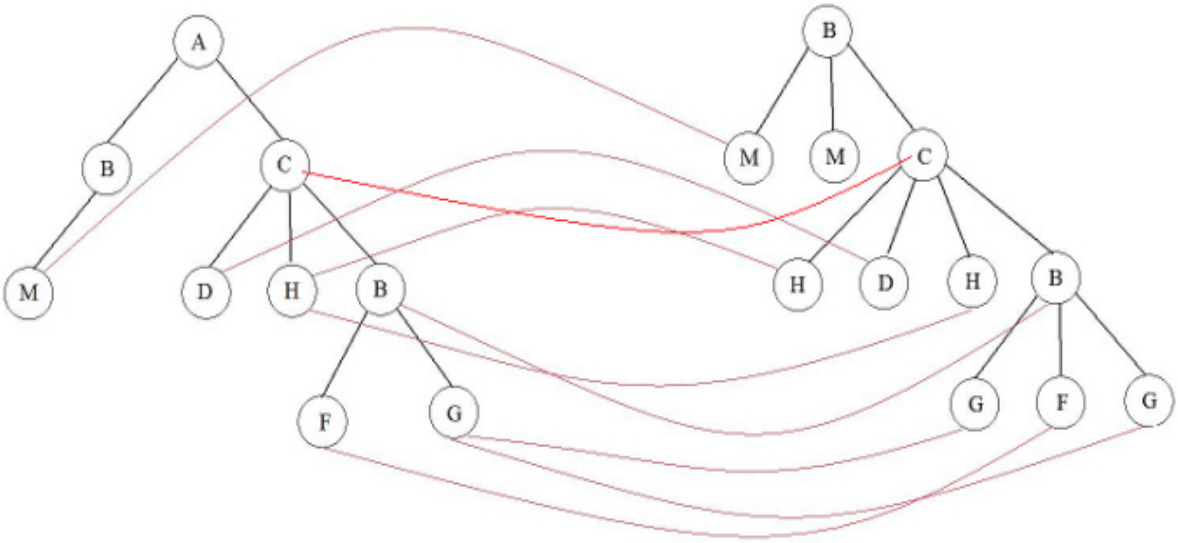

Figure 3-12 : Mapping subtrees using the 'set' criterion

The set criterion subsumes the other criteria since it also ignore the order of calls. The order of call subsumes the identity matching criterion. Figure 3.13 shows this relationship.

The set criterion is the least restrictive and it is expected to result in high similarity, whereas the identity criterion is the most restrictive.

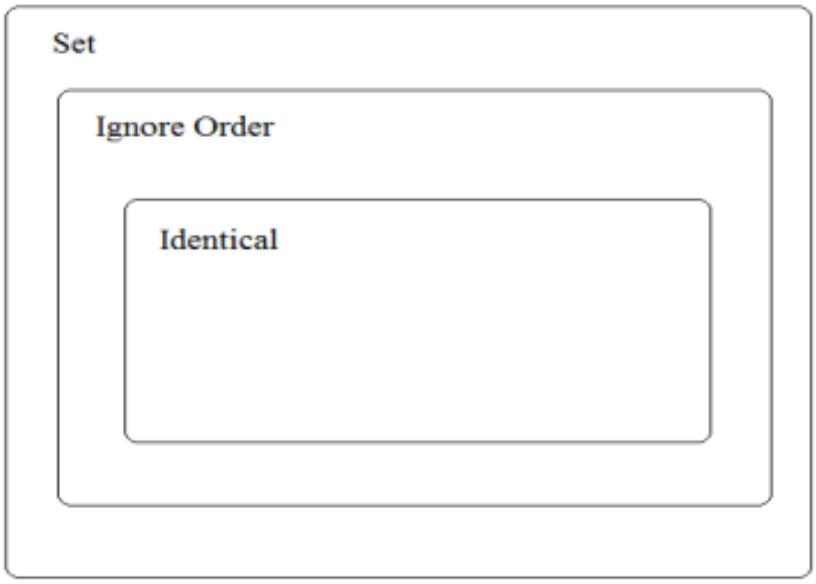

Figure 3-13 : The containment relationship between the matching criteria

3.3.4 The Similarity Metric

Once the mapping set is defined as shown in the previous section. We need to define the sets I and J. This is relatively easy since the set I represents all the subtrees in T1 that are not in T2 and the set J represents the subtrees in T2 that are not in T1. A simple browsing of the trace can determine this. In addition to this, we also need to determine the weight of substitution, insertion, and deletion operations in order to compute the dist(T1, T2), which is defined in Formula 3.18 as:

$$dist(T_1, T_2) = |M|p + |I|q + |J|r \qquad (3.20)$$

Since we do not perform any substitution in our approach then p = 0. This is because we use the mapping set to keep what is similar between the two traces. For everything else, we either insert or delete to make T1 look like T2. We assume, in this thesis, that the insertion and deletion weights (i.e. the values of q and r) is the same and equals 1.

Since $|I| = |T_1| - |M| \wedge |J| = |T_2| - |M|$ then

$$dist(T_1, T_2) = |M|p + (|T_1| - |M|)q + (|T_2| - |M|)r$$

Assuming that $q = r = 1$, and we know that $p = 0$ the distance between $T1$ and $T2$ is therefore:

$$dist(T_1, T_2) = |T_1| + |T_2| - 2|M| \qquad (3.21)$$

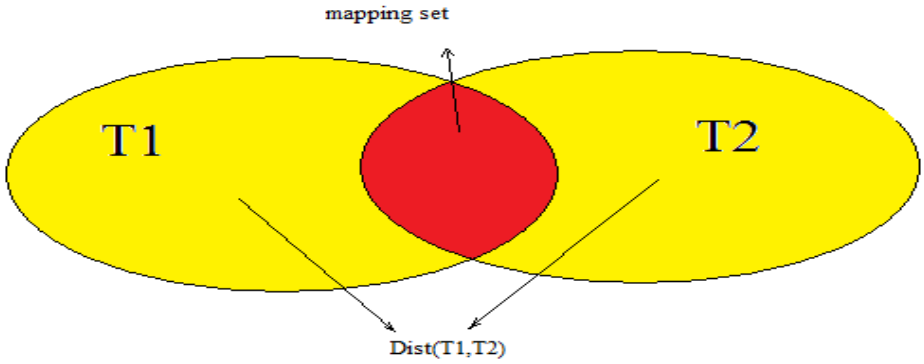

Figure 3-14 : The Venn diagram representing the distance between T1 and T2

In order to find the maximum and minimum value of $dist(T_1, T_2)$ we need to find the maximum and minimum value of $|M|$. The worst case happens when T_1 and T_2 are completely different so $|M| = 0$ (i.e. the mapping set is empty) and therefore $min(|M|) = 0$;

When T_1 is included in T_2 or T_2 is included in T_1 the maximum value of $|M|$ is reached and if $(T_1 = T_2)$ then $dist(T_1, T_2)=0$. In other words, no edit operations are needed to go from T_1 to T_2, and in this case $max(|M|) = min(|T_1|, |T_2|)$

Using Formula 3.21, the following relationships hold:

$$|M| = 0 \Rightarrow dist(T_1, T_2) = |T_1| + |T_2|$$

$$T_1 = T_2 \Rightarrow |M| = |T_1| = |T_2| \Rightarrow dist(T_1, T_2) = 0$$

In order to establish a linear relationship between $dist(T_1, T_2)$ and $dif(T_1, T_2)$ we have provided the graph in figure 3.15 describing the following calculations.

$$dist(T_1, T_2) \propto dif(T_1, T_2)$$

$$tag\ \alpha = \frac{dif}{dist} = \frac{1}{|T|} \Rightarrow dif = \frac{dist}{|T|}$$

$$dist = dif \times |T| \quad (12)$$

$$sim(T_1, T_2) = 1 - dif(T_1, T_2)$$

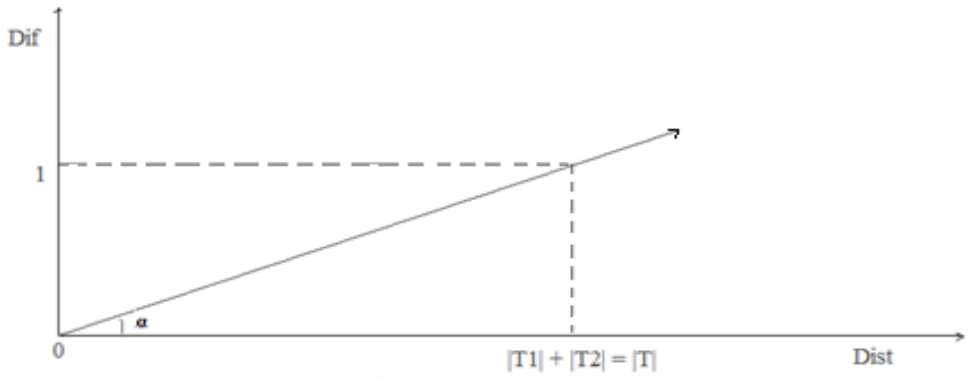

Figure 3-15: *dif* and *dis* are proportionally related

3.4. Summary

In this chapter we presented a complete formal framework to represent concepts that pertain to trace abstraction and correlation. Particularly, we focused on modeling trace structure, its subtrees, and its patterns. A special attention was paid to patterns since they are often used as an effective strategy for trace abstract. We also presented a complete formalization of trace correlation concepts based on the edit distance and pattern matching. The summary of the functions presented in this chapter can be found in Table 3.1.

Table 3.1. Summary of functions and predicates for trace abstraction and correlation framework

Term	Description	Type
|T|	This function returns the size of the trace T	Function
T[i]	The i^{th}node in the Tree T. i is the location of the node	Function
isTrace(u)	Shows if subtree rooted from node u is a proper trace	Predicate
T.root	This function returns the root of the trace T	Function
u.l	This function returns the label of the node u	Function
u.t	This function returns the time stamp of the node u	Function
u.n	This function returns the nesting level of the node u	Function
Parent(u)	This function returns the parent of the node u	Function
Callees(u) or children(u)	This function returns a set of all direct children of node u	Function
subTree(u)	This function returns a set containing a tree rooted from node u	Function
subTree_label	This function is behaving exactly like subTree(), but it returns just label of the nodes	
Path(u)	This function returns a set of all node u's ancestors	Function
fIdentical(u, v)	If two nodes u and v are identical	Predicate
fIgnoreOrder(u, v)	If two nodes u and v are isomorphic under ignore order.	Predicate
fSet(u, v)	If two nodes u and v are isomorphic under set criteria	Predicate
fIdentical_depthLimit(u, v, m)	If two nodes u and v are isomorphic under some criteria and with the limit of the depth up to m	Predicate
fFlattening(u, v)	Ignore the hierarchical structure of subtree and compare them	predicate
sim(u, v)	Returns the similarity of two nodes u and v	Function
dif(u, v)	Return the difference of two nodes u and v	Function
dist(u, v)	Returns the distance of two nodes u and v	Function

Chapter 4. Evaluation

In this chapter, the applicability of our approach is evaluated by comparing different versions of the same system and therefore using the formal framework concepts presented in the previous chapter. We have implemented the framework in Java in the Eclipse environment.

4.1. Target Systems

The traces used in this case study are generated from two Java-based systems (ArgoUML[1] and JHotDraw[2]). We used three traces generated from three versions of ArgoUML and compared them. Similarly, we compared two traces generated from two versions of JHotDraw. The versions used in this thesis are: ArgoUML 0.26, 0.28, and 0.31.5 and JHotDraw 5.2 and 5.3.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a "design exercise" but is quite a powerful tool. Its design relies heavily on some well-known design patterns.

ArgoUML is the leading open source UML modeling tool and includes support for all standard UML diagrams. Both systems have good online documentation. This is particular important to validate our results in the absence of the original designers.

[1] http://argouml.tigris.org/

[2] http://www.jhotdraw.org/

4.2. Generating Traces

We are using TPTP[3] which is an Eclipse plug-in to profile Java applications to instrument both JHotDraw and ArgoUML. Probes have been inserted in each entry and exit of a method. The nesting level is increased as a new function is called and it will be decreased at the return point.

4.2.1. ArgoUML

For each version of ArgoUML, we exercised several features which are: Drawing a UML interface and a class, drawing a generalization dependency between them, and drawing a package that contains two classes which are depended on each other, then switching to the use case diagram mode and drawing an actor connected to two use cases. A trace is generated for every feature (i.e., drawing a class, interface, package, actor and use case).

During the initialization of the program which is just starting and exiting the program there is a trace which is called initialization that has been generated. The traces generated for the above features contain the initialization part. We removed the initialization part from the trace generated from drawing a class. The resulting trace, which is called class_noInit, is used in this thesis to discuss the results obtained. The traces of two versions 0.26 and 0.31.5 are compared to each other and also the traces of the two versions 0.28 and 0.31.5 are compared to each other to show the similarity of different versions of ArgoUML software.

[3] http://www.eclipse.org/articles/Article-TPTP-Profiling-Tool/tptpProfilingArticle.html

The traces are preprocessed before they are compared by removing contiguous repetitions and low-level utilities such as mouse movement events including mouseEntered, mouseExited, mouseReleased, mousePressed and so on.

Table 4.1 shows the statistics about the ArgoUML traces. The table includes the original size of the trace, the size after removing contiguous repetitions and utilities, and the number of patterns detected using the 'identity', 'ignore order', and 'set' criteria.

4.2.2. JHotDraw

To generate traces from JHotDraw, we used the following features: Drawing a circle and rectangle and closing the program. The initialization trace which consists of starting the program and closing it is also generated (in version 5.3 we are opening a new session also). The circle trace contains just the events generated by drawing a circle. Similarly, the rectangle trace contains the events generated when drawing a rectangle. Table 4.2 contains the information about the generated traces.

It should be noticed that in both systems the number of detected patterns using the three matching criteria is very close, which indicates that there is not a significant variation in the patterns of these systems. This shows that additional matching criteria need to be investigated to further generalize patterns. In the following tables sometimes we are using '-' that means a number for this section is not applicable, for example for class_noInit the term "Traces size after removing Utilities" does not mean because this trace is generating after removing utilities and also contiguous repetitions of the traces Class and Initialization, so The size of the trace for the terms "Original trace Size", "trace size after

removing Utilities" and "Trace size after removing Contiguous repetitions" are actually the same.

4.3. Application of the Trace Correlation Technique

We applied It is proved that the similarity metric which is defined in this thesis shows the distance of two routine calls traces when we are mapping the similar patterns to each other. This metric could be valid if it shows the real similarity of two traces, so we need to validate the metric using real data, however validation is not an easy job since similarity could be interpreted optionally, but we are trying to provide the maintainers with the feeling of how different two traces and consequently how different two systems are while we know at least the similarity metrics shows the distance of two routine call trees. Because this metric is capable of comparing two tree structured traces (routine call trace), we are comparing the systems in terms of the methods' name and calling processes (which method calls which).

Table 4.1. Statistics on ArgoUML traces

Trace information \ Feature	Original	Initialization	Class	Package	Interface	Actor	Use Case	Class_noInit
Original Trace Size for 0.26	250152	28052	41584	37134	36777	39859	49132	-
Trace size after removing Utilities for 0.26	249969	28048	41567	37123	36766	39845	49107	-
Trace size after removing Contiguous repetitions and utilities for 0.26	82999	13746	17999	16701	17093	18293	18792	2514
Number of distinct Identical patterns for 0.26	1309	421	595	552	592	620	603	139
Number of all Identical patterns for 0.26	27973	4964	6326	5878	6007	6402	6615	863
Number of distinct Ignored order Patterns for 0.26	1306	421	595	552	592	620	603	139
Number of all Ignored order Patterns for 0.26	27982	4964	6326	5878	6007	6404	6615	863
Number of distinct Set Patterns for 0.26	1301	421	596	552	593	621	603	139
Number of all Set Patterns for 0.26	27991	4966	6331	5880	6013	6411	6619	866
Original Trace Size for 0.28	219790	39524	50074	44606	42545	45882	54302	-
Trace size after removing Utilities for 0.28	219175	39516	50054	44594	42527	45865	54278	-
Trace size after removing Cont. repetitions and utilities for 0.28	82001	16812	21839	19953	20209	21156	21580	2435
Number of distinct Identical patterns for 0.28	1388	460	656	608	642	682	691	152
Number of all Identical patterns for 0.28	31705	6540	8367	7582	7727	8066	8193	868
Number of distinct Ignored order Patterns for 0.28	1392	460	656	608	642	682	691	152
Number of all Ignored order Patterns for 0.28	31726	6540	8367	7582	7727	8066	8193	868
Number of distinct Set Patterns for 0.28	1376	458	654	605	640	680	688	152
Number of all Set Patterns for 0.28	31732	6542	8369	7582	7729	8069	8194	869
Original Trace Size for 0.31.5	390937	49195	61301	61743	50105	57759	60232	-
Trace size after removing Utilities for 0.31.5	389989	49191	61255	61719	50071	57729	60184	-
Trace size after removing Contiguous repetitions and utilities for 0.31.5	125061	18042	23979	21322	22532	23182	24148	3280
Number of distinct Identical patterns for 0.31.5	1533	479	669	605	655	733	766	160
Number of all Identical patterns for 0.31.5	51017	6931	9306	7995	8691	8943	9344	1312
Number of distinct Ignored order Patterns 0.31.5	1531	479	669	605	656	733	766	160
Number of all Ignored order Patterns 0.31.5	51025	6931	9306	7995	8693	8943	9344	1312
Number of distinct Set Patterns For 0.31.5	1519	476	667	602	654	730	763	160
Number of all Set Patterns For 0.31.5	51041	6931	9308	7995	8695	8944	9345	1312

Table 4.2. Statistics on JHotDraw traces

Feature / Trace information	Original	Initialization	Circle	Rectangle	Circle_noInit	Circle_noInit_noRec
Original Trace Size for 5.2	39696	1879	10740	9443	-	-
Trace size after removing Contiguous repetitions for 5.2	1874	635	899	881	148	53
Number of distinct Identical patterns for 5.2	150	54	82	81	22	10
Number of all Identical patterns for 5.2	837	195	299	288	52	21
Number of distinct Ignored order Patterns for 5.2	150	54	82	81	22	10
Number of all Ignored order Patterns for 5.2	837	195	299	288	52	21
Number of distinct Set Patterns for 5.2	148	54	82	81	22	10
Number of all Set Patterns for 5.2	843	195	299	288	52	21
Original Trace Size for 5.3	124505	19800	36701	38817	-	-
Trace size after removing Contiguous repetitions for 5.3	21548	14303	15994	15963	984	260
Number of distinct Identical patterns for 5.3	352	101	213	206	107	29
Number of all Identical patterns for 5.3	8897	5764	6416	6391	464	179
Number of distinct Ignored order Patterns for 5.3	352	101	213	206	107	29
Number of all Ignored order Patterns for 5.3	8897	5764	6416	6391	464	179
Number of distinct Set Patterns for 5.3	339	101	213	206	107	29
Number of all Set Patterns for 5.3	8907	5764	6420	6391	464	179

4.3.1. ArgoUML

Table 4.2 shows the result of applying the trace correlation approach to traces ofArgoUML. We can see that traces of the versions 0.28 and 0.31.5 are more similar than the traces generated from versions 0.26 and 0.31.5. This is justified by the fact that closer versions are, not necessarily, but most of the time more similar.

The initialization phase of different versions is done in a very similar way. For example initialization in versions 0.28 and 0.31.5 are about 90% similar and initialization in versions 0.26 and 0.31.5 are 74% similar. Moreover the Class_noInit in both versions 0.26 and 0.31.5 are 37% similar. This means that the 68% similarity between the traces of the Class scenario of versions 0.26 and 0.31.5 is attributed to the initialization phase. We have not attempted to duplicate this study to all scenarios but we suspect that the high correlation between the various versions is due to the initialization phase which often does not change from version to another.

Table 4.3. Comparison result for Argouml

Feature / Version	Original	Initialization	Class	Package	Interface	Actor	Use Case	Class_noInit
0.26 VS 0.31.5 under Identical Criteria	**0.5165**	**0.7436**	**0.6793**	**0.7316**	**0.7041**	**0.7234**	**0.7243**	**0.3731**
0.28 VS 0.31.5 under Identity Criteria	**0.6604**	**0.9066**	**0.8510**	**0.8904**	**0.8663**	**0.8796**	**0.8607**	**0.4157**
0.26 VS 0.31.5 under IgnoreOrder Criteria	**0.5165**	**0.7436**	**0.6793**	**0.7316**	**0.7042**	**0.7235**	**0.7243**	**0.3731**
0.28 VS 0.31.5 under IgnoreOrder Criteria	**0.6626**	**0.9066**	**0.8516**	**0.8904**	**0.8673**	**0.8796**	**0.8614**	**0.4157**
0.26 VS 0.31.5 under Set Criteria	**0.5166**	**0.7436**	**0.6793**	**0.7316**	**0.7042**	**0.7236**	**0.7244**	**0.3731**
0.28 VS 0.31.5 under Set Criteria	**0.6647**	**0.9070**	**0.8526**	**0.8914**	**0.8679**	**0.8803**	**0.8614**	**0.4227**

4.3.2. JHotDraw

Table 5.6 shows the result of comparing JHotDraw traces. We can see that the difference between the two versions of JHotDraw is important, even when we compare the size of the corresponding traces to each other. For example the size of the original trace after removing contiguous repetitions for version 5.2 is 1,874 while the size of the same trace of the same scenario after removing contiguous repetitions for version 5.3 is 21,548. This shows that significant changes were made to the new version of JHotDraw.

Table 4.3 shows that the two JHotDraw original traces are significantly different (only 10% similarity). We can also see that the matching criteria did not have any effect on the correlation process. This is a clear indication that additional criteria should be used.

Similar to ArgoUML, most of the initialization phase is the same in both versions. However, the implementation of the core functionality (for example the circle with no initialization) is different from one version to another (only 24% similarity). to be are However this is not a good justification why these two versions are very different (similarity of about 10%).

To validate the accuracy of similarity metric we present the analysis of the 'drawing a circle' scenario. First the initialization trace is removed from drawing a circle. Circle_noInit of version 5.2 and rectangle_noInit of version 5.2 are 64% similar and also circle_noInit of version 5.3 and rectangle_noInit of version 5.3 are 71% similar. In order to focus more on core functionality of drawing a circle, we can remove the rectangle from the circle since drawing a circle and rectangle in both versions have many common patterns. Table 4.6 shows the final size of the drawing a circle from which initialization and rectangle are removed. Most of the classes invoked in circle_noInit_noRec are from

EllipseFigure class and we know that this class is responsible for drawing a circle (we checked it in documentation).

Table 4.4. Comparison result for JHotDraw

Feature / Version	Original	Initialization	Circle	Rectangle	Circle_noInit	Circle_noInit _noRec
5.2 VS 5.3 under Identical Criterion	0.1087	0.449	0.0666	0.0653	0.1713	0.2364
5.2 VS 5.3 under IgnoreOrder Criterion	0.1087	0.449	0.0666	0.0653	0.1713	0.2364
5. VS 5.3 under Set Criterion	0.1087	0.449	0.0666	0.0653	0. 21713	0.2364

Table 4.5. Remained distinct methods in circle_noInit_noRec in ver. 5.2

1. CH.ifa.draw.standard.StandardDrawingView.paint
2. CH.ifa.draw.standard.StandardDrawingView.mousePressed
3. CH.ifa.draw.standard.StandardDrawingView.mouseReleased
4. CH.ifa.draw.standard.StandardDrawingView.mouseDragged

Table 4.6. Remained distinct methods in circle_noInit_noRec in ver. 5.3

1. CH.ifa.draw.standard.AbstractFigure.containsPoint
2. CH.ifa.draw.standard.QuadTree.add
3. CH.ifa.draw.standard.StandardDrawingView$1.mouseReleased
4. CH.ifa.draw.standard.SelectionTool.mouseMove
5. CH.ifa.draw.standard.StandardDrawingView$1.mousePressed
6. CH.ifa.draw.util.UndoableTool.mouseDrag
7. CH.ifa.draw.standard.CompositeFigure._addToQuadTree
8. CH.ifa.draw.util.UndoableTool.mouseDown

Table 4.5 and 4.6 show the list of distinct method in circle_noIit_noRec in versions 5.2 and 5.3 respectively from which common distinct methods are removed. An analysis of these two versions with a particular emphasis on drawing the circle shows several major differences in the implementation of this scenario from one version to another, which justifies the low similarity metric obtained by our approach. In what follows, we show the key differences:

- The method “StandardDrawingView.paint” in ver. 5.2 is changed to the “StandardDrawingView.paintComponent”, and both of these methods are responsible for painting the drawing view and both are calling the “painter.draw” method.

- In addition, the methods “StandardDrawingView.mousePressed”, “StandardDrawingView.mouseReleased” and “StandardDrawingView.mouseDragged” have been defined directly inside the class “standard.StandardDrawingView” in ver. 5.2 but these methods are defined indirectly inside the “standard.StandardDrawingView” in ver. 5.3. The methods “mousePressed” and “mouseReleased” will be defined when “MouseListener”object “ m1” is created and the method “mouseDragged” will be defined when “MouseMotionListener” object “mm1” is created in 5.3. Although the “StandardDrawingView.mouseMoved” is not included in our distinct method of circle_noInit_noRec (it is removed when the rectangle trace is removed from the circle trace), “StandardDrawingView.mouseMoved” is defined indirectly when “mm1” object is created in 5.3 while it is defined directly inside the class “StandardDrawingView.mouseDragged” in 5.2.

- Although “standard.AbstractFigure.containsPoint” is implementd in ver. 5.2 but in the scenario ‘drawing a circle’ it is not called. In ver. 5.3 “standard.SelectionTool.mouseMove” is an ancestor of “standard.AbstractFigure.containsPoint” which is not implemented in version 5.2.

- In version 5.2, the classes “standard.QuadTree”, “util.UndoableTool” and “contrib.DragNDropTool” are not implementd and these are new classes in the version 5.3. Also the methods “CompositeFigure._addToQuadTree”, “CompositeFigure._removeFromQuadTree” and “StandardDrawingView.mouseMoved” are not implemented in 5.2.

- The method “CompositeFigure.findFigure” and “DecoratorFigure.containsPoint” are implemented but they are not invoked in the scenario drawing a circle in 5.2. In 5.3 “DragNDropTool.setCursor” which is an ancestor of “CompositeFigure.findFigure” and “DecoratorFigure.containsPoint” is not implemented in 5.2 since the class “contrib.DragNDropTool” is not implementd. The method “CompositeFigure.figureChanged” is declared in 5.2 but there is no implementation for this method.

4.4. Discussion

In this chapter, we showed the applicability of our formal framework by comparing traces generated from different versions of two systems. We showed that using our approach, we can perform powerful analysis of system behavior such as comparing differences in subsequent versions of the same software system. This type of analysis opens the door to new techniques and tools that enable effective analysis of large traces.

We also validated the trace correlation process itself which is based on behavioral patterns. However, it appears that the matching criteria we have formalized so far are not

sufficient since they do not provide significant difference. Therefore, there is a need to investigate further techniques for matching subtrees to further generalize patterns.

Chapter 5. Conclusion

5.1. Research Contributions

Our main contribution in this thesis is a formal framework for trace abstraction and correlation. Trace abstraction is needed to simplify the analysis of large traces and hence enable the understanding of the behavioural aspect of software.

We showed how traces of routine calls and related concepts could be formalized to enable effective development of analysis techniques. We also presented a complete set of formal methods that formalize traces abstraction techniques based on pattern matching. Another important contribution of this thesis is the formal representation of matching criteria. We believe that this can facilitate analyzing comparing techniques based on these matching criteria.

We also presented an approach for trace correlation. Our approach describes how traces generated from different systems can be compared by investigating the common patterns they contain. For this, we used the edit distance and proposed a way to match subtrees based on matching criteria. The formal framework is implemented in Java in the Eclipse environment.

Finally, the last contribution of this thesis consists of an application of the formal framework with an emphasis on the trace correlation process. We applied our approach to several traces generated from two different object-oriented systems. We showed how our approach permits the analysis of traces and detecting similarities and differences.

5.2. Opportunities for Further Research

The next immediate future work is to continue the formalization process. For example, we can formalize other matching criteria that were not covered in the thesis.

There are also several other research directions. We need to build on the top of the framework analysis techniques where proofs of their effectiveness can be established using the concepts (with improvements) presented in our framework. The framework needs also to be extended to other trace abstraction techniques such as sampling, grouping and etc.

The definition and implementation in our framework are not the most optimized ones and one of the future works could be to study the optimal implementation of the concepts.

We discussed the need for a trace correlation in this thesis and its help for program comprehension. The correlation metric in this thesis is calculated based on the distance of two traces with respect to similar patterns in two traces, and it does not mean that we need to use the distance of two traces to obtain the similarity metric, we can use other ways to get the correlation metric but we need to find a reasonable way to validate it.

Finally, the concepts presented in this thesis should be integrated with trace analysis tools and serve as the main mechanism on which other techniques can be build.

5.3. Closing Remarks

Trace analysis is a difficult topic due to the overwhelming information generated from even a small system but if it is done properly, it can yield powerful tools that permit the analysis of system behavior. In this thesis, we presented a formal framework for trace abstraction and correlation, two key activities that can facilitate greatly the analysis of traces. We hope that our framework can advance this area where the challenges are enormous but, if overcome, the benefits are rewarding.

Bibliography

Parnas 94 — D. L. Parnas, “Software Aging”, *In Proc. of the 16th International Conference on Software Engineering,* pp. 279-287, 1994.

IEEE93 — IEEE std. 1219: Standard for Software Maintenance. Los Alamitos CA., USA. IEEE Computer society Press, 1993.

Lientz 80 — B. P. Lientz and E. B. Swanson. Software Maintenance Management. Addison Wesley, 1980.

Bennett 00 — K. H. Bennett , V. T. Rajlich, “Software maintenance and evolution: a roadmap”, *In Proc. of the Conference on the Future of Software Engineering*, pp.73-87, 2000.

Rugaber 95 — S. Rugaber, “Program comprehension”, In Encyclopaedia of Computer Science and Technology, 35(20), pp 341-368, 1995.

Fjeldstad 83 — K. Fjeldstad and W. T. Hamlen., ‘‘Application Program Maintenance Study: Report to Our Respondents’’, *In Proc. of GUIDE 48, The GUIDE Corporation, Philadelphia,* pp. 13-30, 1983.

Mayrhauser 95 — A. V. Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer, 28(8),* pp. 44-55, 1995.

Hamou-Lhadj 06 A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 12th International Conference on Program Comprehension,* pp. 181-190, 2006.

Hamou-Lhadj 03(a) Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge, " Techniques for Reducing the Complexity of Object Oriented Execution Traces", *In Proc. of the 2nd Annual Designfest on Visualizing Software for Understanding and Analysis,* pp. 45-30, 2003.

Zayour 00 Zayour and T.C. Lethbridge, "A Cognitive and User Centric Based Approach For Reverse Engineering Tool Design", *CASCON*, 2000.

Hamou-Lhadj03(b) Hamou-Lhadj, A. and Lethbridge, T.C., "An efficient algorithm for detecting patterns in traces of procedure calls", *In proc. WODA 2003 ICSE Workshop on Dynamic Analysis* pp. 33-36, 2003.

De Pauw 98 W. De Pauw, D. Lorenz, J. Vlissides and M. Wegman, "Execution Patterns in Object-Oriented Visualization", *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98), USENIX,* 1998, pp. 219-234.

Tai 79 K. C. Tai, "The tree-to-tree correction problem",*ACM*, 26(3):pp. 422-433, 1979.

Kuhn 06 Kuhn, A. and Greevy, O., “Exploiting the analogy between traces and signal processing”, *22nd IEEE International Conference on Software Maintenance (ICSM'06),* 2006.

A.Hamou-Lhadj 04 A. Hamou-Lhadj, and T. C. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, CASCON 2004, IBM Press, ACM Digital Library , Toronto, Canada, pp. 42-54, 2004.

De Pauw 93 De Pauw W., Helm R., Kimelman D., and Vlissides J., “Visualizing the Behaviour of Object-Oriented Systems”, In Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages,and Applications, ACM Press, pages 326-337, 1993

Jiang 95 T. Jiang, L. Wang, and K. Zhang. “Alignment of trees—an alternative to tree edit”. *Theor. Comput. Sci.*, 143(1), pp. 137–148, 1995.

Selkow 77 S. M. Selkow. “The tree-to-tree editing problem”. *Inform.Process. Lett.*, 6(6), pp. 184–186, 1977.

Tanaka 88 E. Tanaka and K. Tanaka. “The tree-to-tree editing problem”. *Int. J. Pattern Recogn. and Artif. Intell.*, 2(2),pp. 221–240, 1988.

Valiente 01(a) G. Valiente. “Simple and efficient tree comparison”. Technical Report LSI-01-1-R, Technical University of Catalonia, Department of Software, 2001.

Yang 91	W. Yang. Identifying syntactic differences between two programs. *Software—Practice and Experience*, 21(7), pp.739–755, 1991.
Valiente 01(b)	Valiente, G.," An efficient bottom-up distance between trees", *In proc. 8th International Symposium on String Processing and Information Retrieval*, pp. 212-219, 2001.
Gusfield 97	D. Gusfield. "*Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology"*. Cambridge University Press, 1997.
Stephen 94	G. A. Stephen. *String Searching Algorithms*. World Scientific Press, 1994.
Chan 03	Chan A, Holmes R, Murphy GC, Ying ATT. "Scaling an object-oriented system execution visualizer through sampling". *In Proc. 11th Int. Workshop on Program Comprehension (IWPC), IEEE*, 2003; 237–244.
Whaley 00	Whaley, J., "A portable sampling-based profiler for Java virtual machines", *In Proc. of the ACM 2000 conference on Java Grande,* pp. 78-87, 2000.
Dugerdil 07	Dugerdil, P., "Using trace sampling techniques to identify dynamic clusters of classes", *In proc. 2007 conference of the center for advanced studies on Collaborative research,* pp. 306—314, 2007.

Valiente 00 G. Valiente. “Simple and efficient subtree isomorphism”.Technical Report LSI-00-72-R, Technical University of Catalonia,Department of Software, 2000.

Hamou-Lhadj 02 Hamou-Lhadj, A. and Lethbridge, T.C., “Compression Techniques to simplify the Analysis of Large Execution Traces”, *In proc. IWPC’02, 10th International Workshop on Program Comprehension* pp. 159-168, 2002.